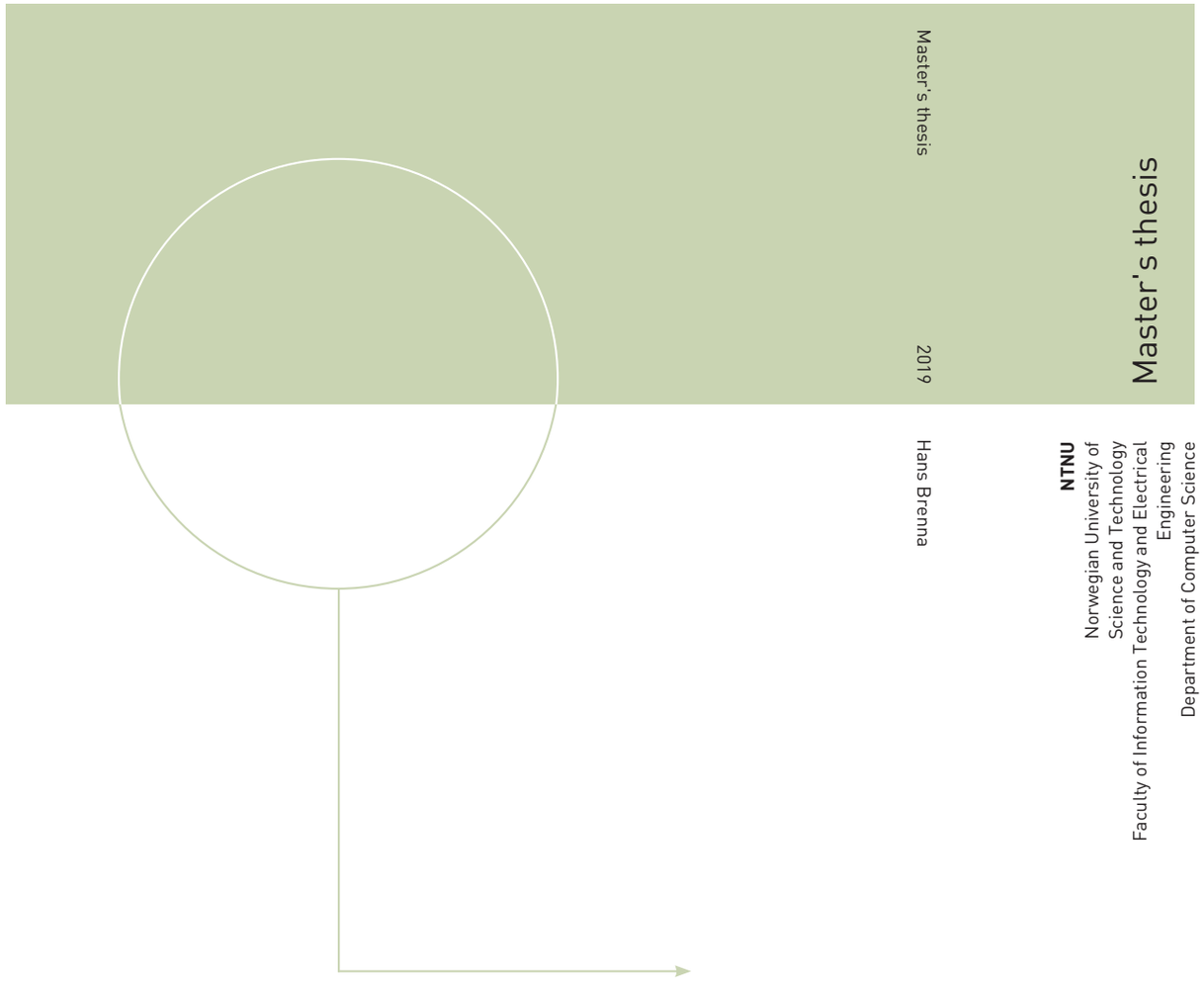


Master's thesis

2019

Hans Brenna

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Hans Brenna

Speeding up mesh descriptors by implementing them on the GPU

May 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU

Norwegian University of
Science and Technology

Speeding up mesh descriptors by implementing them on the GPU

Informatics
Submission date: May 2019
Supervisor: Theoharis Theoharis
Co-supervisor: Bart Iver van Blokland.

Norwegian University of Science and Technology
Department of Computer Science

Summary

3D feature descriptors are capable of providing point-to-point correspondence on different surfaces. They are therefore used in a variety of different applications within visual computing. However, these applications often find the computation of feature descriptors to be a bottleneck for their performance. A key observation is that these feature descriptors can be computed independently of each other, and are therefore good candidates for parallel computation. Modern GPUs, together with specialized APIs, can be utilised for such highly parallel problems. This thesis has examined various GPU implementations of four popular feature descriptors: Spin Images, Point Feature Histograms, Signature of Histograms of Orientations and Fast Point Feature Histograms. As a result, the thesis suggest different approaches to GPU based feature descriptors and applicable GPU optimizations.

3D egenskapsbeskrivelser er en måte å skape punkt-til-punkt korrespondanse mellom forskjellige overflater. De brukes derfor i en rekke applikasjoner innenfor visuell datababehandling. Ytelsen til disse applikasjonene er ofte begrenset av hvor fort egenskapsbeskrivelser kan beregnes. Det viser seg imidlertid at egenskapsbeskrivelser kan beregnes uavhengig av hverandre. De er derfor godt egnet til parallel beregning. Moderne grafikkprosessorer, sammen med spesialiserte APIer, kan utnyttes for parallel beregninger av denne typen. Denne oppgaven har sett på GPU implementasjoner av fire populære egenskapsbeskrivelser: Spin Images, Point Feature Histograms, Signature of Histograms of Orientations og Fast Point Feature Histograms. Dette har resultert i forslag for hvordan algortimer for egenskapsbeskrivelser kan implementers på grafikkprossesorer, og hvordan disse algortimene kan optimaliseres.

Contents

Chapter 1

Introduction

In the field of computer vision, 3D feature descriptors are used in shape retrieval, object recognition and modelling. The purpose of a feature descriptor is to describe the neighborhood of a surface vertex in such a way that vertices of different surface may be compared against one another. A good feature descriptor is characterized as being robust, descriptive and compact (Guo et al., 2016). Robustness means that the feature descriptor is insensitive to noise, clutter and occlusion which might be attributed to sampling variations. A descriptor is said to be descriptive if it manages to encompass the geometric information of the underlying surface. Lastly the compactness is a measure on how much each value in a computed feature descriptor contributes towards its performance(Guo et al., 2016). Throughout this thesis the term feature extraction refers to the process of using a particular algorithm in order to create a descriptor of a point.

Modern graphical processing units (GPU) are capable of solving highly problems, despite the problems not being inherently related to graphics rendering. Frameworks such as CUDA and OpenCL enables developers to utilize the GPU for computation that would traditionally be done on the CPU. This is accomplished by invoking kernel functions that are executed in massively parallel fashion on the GPU.

Section 1.1 details the motivation for computing feature descriptors using the GPU. Section 1.2 outlines the research questions for the thesis.

1.1 Motivation

Feature descriptors are a vital part of the pipeline in many tasks related to computer vision. However, the feature extraction often becomes the bottleneck of such tasks. (Palossi et al., 2013). A key observation is that each descriptor can be computed independent of

the others. Because of this, feature description algorithms are well suited for GPU implementation (Hu and Nooshabadi, 2015). Several feature descriptors have already been implemented on the GPU. Some of them providing runtime speeds 40 times faster than on the CPU. (Hu and Nooshabadi, 2015).

Other algorithms, not related to feature descriptors, have also been implemented on the GPU. Well known examples includes, but are not limited to, N-Body, Reduction and Scan (Nylons, 2007)(Wilt, 2013). By applying knowledge of the GPU hardware, it is possible to apply optimizations in order to make them run faster (Wilt, 2013). Extensive research has been done on how to optimize the mentioned algorithms not related to feature descriptors. While there exists several GPU implementations of feature descriptors, there exists little research comparing the degree to which these methods are able to utilise the available computational power of the GPU, both by the algorithm's design and their actual implementation. By determining the best characteristics, faster feature extraction implementations could be developed for the GPU. This in turn could alleviate the bottleneck where feature extraction has previously incurred a relatively high computational expense, in some cases allowing for faster visual computing applications.

1.2 Research Questions

The work done in this thesis aims to answer the following:

Research Question: *How can efficient computation of feature descriptors be implemented on the GPU?* Implementations are considered to be efficient if they exhibit the following characteristics. The algorithms should be designed with parallel execution in mind. They should also be optimized by leveraging the GPU architecture. To this end, these follow-up questions are derived:

RQ1: *How have CPU based feature descriptor algorithms been implemented on the GPU?*

RQ2: *What measures can be taken in order to optimize them with regards to utilisation of the GPU?*

Chapter 2

Background

This chapter supplies the theoretical background required in order to answer the research question. Section 2.1 explains a selection of feature descriptors and the steps related to computing feature descriptors in general. It also gives a detailed explanation of the following feature descriptors: Signature of Unique Histograms of Orientations (SHOT), Spin Images (SI), Point Feature Histograms (PFH) and Fast Point Feature Histograms. This thesis concerns itself with algorithms written for the GPU. Therefore section **??** introduces parallel programming using CUDA.

2.1 Feature Descriptors

The computation of feature descriptors generally consists of the following steps: First, 3D data along with properties such as normal information is acquired.Then one selects which vertices to compute feature descriptors for. Any such vertex will, for the sake of disambiguation, be referred to as a feature vertex.

The next step is to identify the neighborhood of any feature vertex. Ideally the neighborhood should contain, or be equivalent to the *support region* of the feature vertex. The support region can be thought of as a geometric primitive, such as a sphere or cylinder. This primitive is then centered around the feature point. Only neighbor vertices contained within the support region will contribute in computing a descriptor for the feature vertex.

Once the support region for a feature vertex has been defined, its feature descriptor can be computed. Feature descriptors creates a function that maps vertices within the support to a vector of values. By comparing these vectors, one can discern the similarity between surface points. (Guo et al., 2016) provides a benchmark comparison of popular feature descriptors. Among these, SHOT, SI, PFH and FPFH have been implemented on the

GPU. With the exception of SHOT, all of these had source code that was available to the author. GPU implementations of these descriptors have also been examined in an academic setting, with the exception of PFH. As a result, this thesis examines these four feature descriptors and their GPU implementations in detail.

The first subsection looks at how neighborhoods can be efficiently determined, the second subsection explains how the selected types of feature descriptors are extracted.

2.1.1 Nearest neighbor searches

As mentioned earlier, nearest neighbor searching must be done for every feature descriptor that is computed. Therefore the impact of this step should be examined. This section aims to explain how neighbor searches can be sped up using data structures known as K-D trees and Octrees.

Octrees and K-D Trees

The aim of both octrees and k-D trees is to provide a form of spatial partitioning. They do this using slightly different approaches.

The steps required to construct an octree are the following: First,the node at the top level of the tree contains all vertices one wishes to partition. A plane is then placed for each of the X,Y and Z-axis. Each plane is orthogonal to its respective axis, and contains the center point of the node. These planes then together splits the node into eight equally sized children. The same procedure is then performed recursively for every child node.

Whether or not a node should be divided or treated as a leaf-node depends on the break conditions. These conditions are usually related to the depth of the node in the tree, or the number of vertices contained within. In a k-d tree, each parent node has two children.

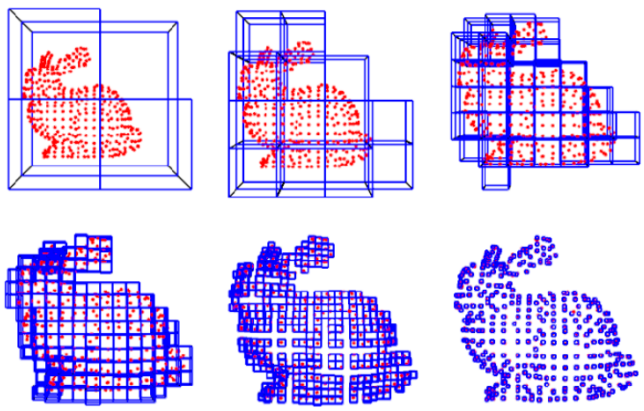


Figure 2.1: Octree partitioning of the Stanford Bunny model (Kammerl et al., 2012)

Each node is split along one of its *dimensions* by a threshold value. In the context of 3D models, the term dimension refers to either the X, Y or Z axis of the model. A different dimension is selected for each level of the tree. It is common to select dimensions in order of descending dimension variance. Another approach is to select dimensions in order of descending dimension length (Silpa-Anan and Hartley, 2008). The threshold, or split, value used to partition items among the child nodes must be determined. This can be the middle of the splitting dimension, however this might lead to children not containing items (Elseberg et al., 2012). A different approach is to observe the items of the current node, and use the median value of the splitting dimension among these items. Figure 2.2 shows a k-d tree partitioning with the first split marked by a red plane, the second split marked by a green plane and the third split marked by a blue plane. By applying

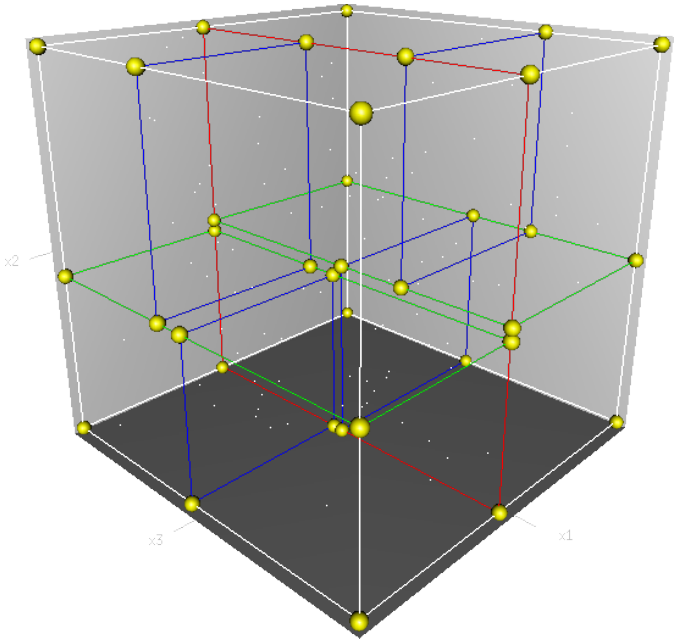


Figure 2.2: K-D Tree Partitioning,

one of the aforementioned spatial structures, efficient nearest neighborhood searches can be implemented reducing complexity from *(O(nm)* down to *O(n log m)* Elseberg et al. (2012)

2.1.2 Signature of Histograms of Orientations

Signature of Histograms of Orientations (SHOT) is a feature descriptor that provides a description of the underlying neighborhood by employing a spherical partitioning scheme of its support region. This results in several volumes, each with their own local histogram (Tombari et al., 2010). The total histogram size is the product of the number of local

histogram bins, and the number of local histograms. In the context of computer programming, this can be described as a 2D array. Each row index corresponds to a volume's local histogram, and the column index would represent the local histogram bin. Figure 2.3 shows the support volume divided using two radial, two elevation and four azimuth divisions. This results in sixteen local histograms.

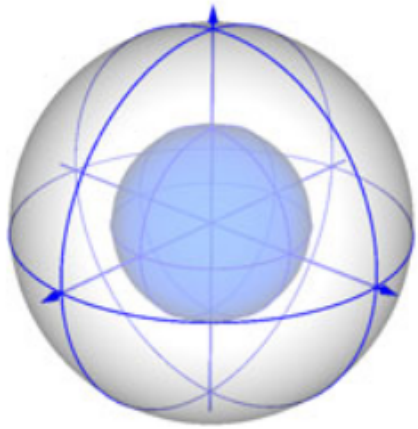


Figure 2.3: SHOT sub-volumes (Tombari et al., 2010)

A local reference frame (LRF) is created for every SHOT descriptor, so that it remains unique and invariant to scaling and translation of the model. The LRF is created by first assembling the covariance matrix C of the sum of all vertices in the support region and performing an Eigenvalue decomposition on this matrix. Each of the X,Y and Z-axis of the LRF are assigned to the Eigenvectors of C. In order to disambiguate the sign of the axes, the technique in (Bro et al., 2008) is applied.

After the LRF has been constructed, initial bin values are assigned to every vertex within the support. Given a vertex and its associated normal N, the initial bin value is given by equation 2.1

$$\begin{aligned} Dot\ Product &= L\vec{R}F_Z \cdot \vec{N} \\ Dot\ Product &= [-1, 1] \\ Initial\ Bin\ Value &= \frac{Dot\ Product + 1}{2} \cdot number\ of\ bins \end{aligned} \tag{2.1}$$

The same vertex then needs to be mapped to a histogram bin in one of the support sphere's local histograms. This means that both the volume index (row index) and the local histogram index (column index) must be computed. Thus its position within the spherical partitioning must be determined. First, the coordinates of the vertex are translated to the local ones within the LRF. Based on the local coordinates of the vertex, its volume index can be computed. The equation used for determining this index depends on the exact number divisions of the spherical support region.

After computing the global index for the vertex, its contribution is smoothed across the histogram. This is done by performing a quadrilinear interpolation with respect to the adjacent volumes in the spherical support region, and adjacent bins in the local histogram. The output of the SHOT descriptor is the concatenation of all its local histograms.

2.1.3 Point Feature Histograms

Point feature histogram (PFH) descriptors are based on how points in a spherical support region are placed relative to each other (Rusu et al., 2008). The steps to compute this descriptor are as follows: First the members of the support region are identified by means of a nearest neighbor search, limited by a radius r. Then every possible pair of vertices $(P_i, P_j) \in Support(V_{feature})$ is processed. For each pair, a source vertex P_s and target vertex P_t are selected, such that the angle between the line connecting the points and the normal of P_s is as small as possible. Figure 2.4 shows that he angle β between N_i and the connecting line is the smallest. Vertex P_i will be therefore be chosen as the source vertex.

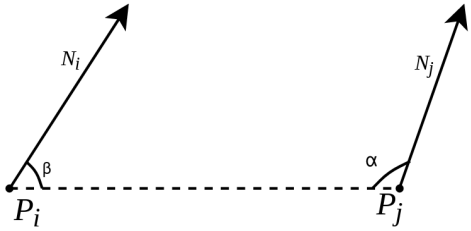


Figure 2.4: Source and Target Vertex

Using the vertices from the pair and their respective normals a Darboux frame is created for every pair. The axes of the frame, represented by vectors u, v, w is defined by equation 2.2.

$$\begin{aligned} u =& n_s \\ v =& (P_t - Ps) \times u \\ w =& u \times v \end{aligned} \tag{2.2}$$

The output of the PFH algorithm is a histogram, where every pair of neighbor vertices contributes to one of the histogram bins.To select the bin, the functions given in equation 2.3 are computed. Henceforth, these functions will be referred to as *pair features*.

$$\begin{aligned} f_1 &= \vec{n_t} \cdot \vec{v} \\ f_2 &= ||P_t - P_s|| \\ f_3 &= \frac{\vec{u} \cdot (P_t - P_s)}{f_2} \\ f_3 &= atan(\vec{w} \cdot \vec{n_t}, \vec{v} \cdot \vec{n_t}) \end{aligned} \tag{2.3}$$

Since all dot products are computed using normalized vectors, the definition interval of f_1 and f_3 is $[-1, 1]$. The definition interval of f_2 is $[0, r]$, and f_4 has the definition interval of $[-\pi, \pi]$.

The pair features then undergoes discretization. For every function f_i a step size $step_i$ is supplied. The definition interval of each function is then divided into b_j equally sized intervals by the step size. The discrete value df_i is given by equation 2.4.

$$\begin{aligned} Offset_i &= \begin{cases} \frac{step_i \cdot b_i}{2}, & \text{if } i \neq 2 \\ 0, & \text{otherwise} \end{cases} \\ df_i &= \begin{cases} \left\lfloor \frac{f_i + offset}{step_i} \right\rfloor, & \text{if } f_i + Offset_i < b_i \cdot step_i \\ b_i - 1, & \text{otherwise} \end{cases} \end{aligned} \tag{2.4}$$

The bin that the pair should contribute to is given by equation 2.5.

$$bin = df_1 + \sum_{k=2}^{k=i} \left(df_k \prod_{j=1}^{k-1} b_j \right) \tag{2.5}$$

Once the the bin is determined, the corresponding value in the output is incremented. The exact increment is defined by equation 2.6

$$Increment = \frac{100}{\textit{Number of processed pairs}} \tag{2.6}$$

2.1.4 Fast point feature histograms

Closely related to PFH, fast point feature histograms trades of some of its descriptiveness in exchange for faster computation (Rusu et al., 2009). The FPFH descriptor is built upon another one, namely Simplified Point Feature Histograms (SPFH). Thus computing FPFH descriptors is a two step process.

The first step is computing SPFH descriptors for every vertex in the model. As in PFH computation, a support region and its members are determined using a nearest neighbors search. Every neighbor vertex forms a vertex pair with the feature vertex. Using the same technique as in PFH computation, discrete pair features are created for all vertex pairs. In PFH, the Euclidean distance between vertices is one of the pair features. However, this is not the case for FPFH descriptors. The reason is that the Euclidean distance has little impact on the robustness of the descriptor (Rusu et al., 2009).

Unlike the PFH algorithm, the pair features are not correlated to each other. In other words, each pair feature f_i has its own histogram with b_i bins. This fact reduces the final size of the histogram from $\prod_{i=1}^{i} b_i$ to $\sum_{i=1}^{i} b_i$. Each discrete feature value represents which bin in the histogram to be incremented. While PFH descriptors consists of one large histogram, FPFH is the concatenation of several smaller ones, each representing one of the pair features.

Once SPFH descriptors have been computed, the next step is to compute FPFH descriptors. To compute the FPFH of a feature vertex, a spherical support and its members are determined as before. Then, the weighted sum of the neighbours' SPFH is computed using equation 2.7. Here p_k is a member of the support, and w_k is the distance between the feature vertex and p_k.

$$FPFH(fp) = SPFH(fp) + \frac{1}{k} = \sum_{i=1}^{k} \frac{1}{w_k} SPFH(p_k) \tag{2.7}$$

.

Once the weighted sum has been computed, the final histogram is normalized. While the complexity of the PFH algorithm is $O(n \cdot k^2)$, FPFH reduces this to $O(n \cdot k$ Where k is the maximum number of members in a support region and n is the number of descriptors. Additionally the size of the descriptor is reduced.

2.1.5 Spin Image

The Spin Image (Hebert and Johnson, 1999) descriptor represents a cylindrical support region containing 3-D vertices as a 2-D image. This requires a function that translates from 3-D to 2D. First a 2D coordinate system is defined. The first axis is the line containing the normal of the feature vertex. The second axis is the distance from this line. We call these axes the β-axis and the α-axis respectively. Vertices whose β and α coordinates exceeds the user-defined maximum values will not be evaluated. These maximum values can be

visualized as the height and radius of the cylindrical support volume, or the width and height of the 2D image.

For every member of the support, their α and β coordinates are given by equation 2.8.

$$\begin{aligned} \beta &= ||\vec{D}|| \cdot cos\theta \\ \alpha &= ||\vec{D}|| \cdot (1 - cos\theta^2) \end{aligned} \tag{2.8}$$

Here $\vec{D}$ represents the vector between the feature point and the support member. The angle θ is the angle between $\vec{D}$ and normal of the feature point. By defining a step size,

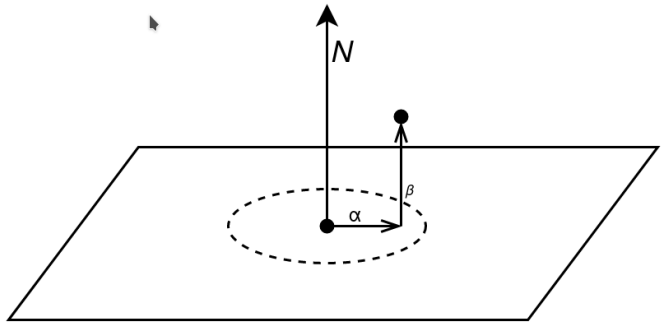


Figure 2.5: The feature point along with its normal N

or pixel size, s, the axes of the support volume is divided into b bins. After (β and α have been calculated, they need to be mapped to pixel coordinates i and j. This is done using equation 2.9 where W.

$$i = \left\lfloor \frac{\beta + \frac{W}{2}}{s} \right\rfloor \, , \; j = \left\lfloor \frac{\alpha}{s} \right\rfloor \tag{2.9}$$

Once the member vertex has been mapped to the image, its contribution is smoothed across four adjacent bins in the image using bilinear interpolation. The interpolated weights A and B, as well as the increments of spin image bins are computed as in equation 2.10.

$$\begin{aligned} A = \alpha - j \cdot s \; , \; B = \beta - i \cdot s \\ SI(i,j) &= SI(i,j) + (1-A) * (1-B) \\ SI(i+1,j) &= SI(i+1,j) + (A) * (1-B) \\ SI(i,j+1) &= SI(i,j+1) + (1-A) * (B) \\ SI(i+1,j+1) &= SI(i+1,j+1) + (A) * (B) \end{aligned} \tag{2.10}$$

2.2 Parallel Computing on the GPU

. Several hallmarks of an efficient GPU implementation are listed in section **??**. One of these concerns itself with the design of the algorithm. Therefore, an understanding of how one can design parallel algorithms is required. As such, subsection 2.2.1 introduces the reader to parallel programming. Another hallmark concerns itself with how the implementation in question can leverage the GPU. Therefore the subsequent subsections introduces GPU programming using CUDA and outlines the architecture of the GPU.

2.2.1 Introduction to Parallel Computing

The speed at which a program takes to run, is often bound by the throughput of the processing unit. The throughput of the processing unit is the number of instructions it can process in a given time interval. Modern CPUs have multiple cores that can run simultaneously. This increases the theoretical throughput available. However, to take advantage of this, an algorithm must partition its instructions over multiple *threads*. These threads can then be executed on different cores, at the same time.

When deciding on which instructions to assign which thread, certain considerations must be taken into account. For example, instructions might be dependent on each other, or the workload might become uneven among threads. Threads might even need to communicate to each other during their execution. To tackle these challenges, one should attempt to divide the original problem into several smaller ones. Dependencies between threads might lead to one thread idling while waiting for another thread to catch up. Communication between threads can be complex. Therefore, these considerations must be taken into account when dividing the original workload into sub-problems. Once suitable sub-problems are identified they can be distributed to the available threads.

The design of modern GPUs allows them to be effective for highly parallel problems not related to graphics rendering (Owens et al., 2008). The GPU, or *device*, contains a number of multiprocessors which are all tailored towards a high throughput of instructions. In addition to the processors, the device also has its own dedicated memory known as device memory. In order to write parallel programs that uses the GPU, developers may use either CUDA or OpenCL. CUDA (Nvidia, 2019d) is Nvidia's own proprietary API created for Nvidia GPUs. OpenCL on the other hand is a cross-platform open-source API designed to be used on a multitude of multiprocessor types (Kronos Group, 2019). All but one of the GPU implementations of feature descriptors available to the author, were written using CUDA. Therefore the rest of this thesis will focus on CUDA and Nvidia GPUs. Despite being different APIs, they share several core concepts. As a result, the thesis should contribute to CUDA and OpenCL development alike. Figure 2.6 gives an overview of CUDA and OpenCL terminology.

CUDA	OpenCL
thread	work-item
warp	wavefront
thread block	work-group
grid	computation domain
global memory	global memory
shared memory	local memory
local memory	private memory
streaming multiprocessor (SM)	compute unit
scalar core	processing element

Figure 2.6: Cuda and OpenCL Terminology

2.2.2 GPU Architecture

The problems that are suited for GPU processing are those that can be expressed as data-parallel problems (Nvidia, 2019d). For this type of computation where the same instructions are issued for multiple data in parallel, the need for features such as optimized control flow becomes redundant to a certain degree. With this in mind, it becomes evident that these features can be replaced with ones focused on increasing the compute capacity of the processing unit as illustrated in **Fig 2.7** In order to provide a higher throughput

Figure 2.7: Allocation of transistors on CPU and GPU

than their CPU counterparts Nvidia GPUs introduces multiple Streaming Multiprocessors (SM) organized together as show in figure 2.9. The SM is a multi-core processor specialized for high computation throughput, achieved by having far more cores than a CPU. Each CUDA core is however much slower and much simpler than a typical CPU core. Essentially a CUDA core only provides a mathematical pipeline that can be used for arith-

metic computation. In addition to CUDA cores, there are also special function units (SFU) and units for reading and writing to memory (LD/ST units). Newer architectures have also seen specialized cores being added to the SM, such as tensor cores and ray tracing cores. For the SM to issue instructions to its cores, LD/ST and SFU units, every SM has a number of units related to *warp scheduling*, these are detailed in section 2.2.4. The different types of memory on each SM, are outlined in section 2.2.5. Figure 2.8 shows the SM used in the Pascal architecture.

2.2.3 CUDA Runtime and SIMT

The CUDA runtime enables developers to write applications for the GPU using C/C++ language extensions. In CUDA, functions known as kernels are invoked by the CPU, henceforth known as the *host*, and executed on the device. Upon invocation a number of threads structured in a *thread hierarchy* are made available to the developer. The thread hierarchy consists of a grid composed of blocks which in turn contain several threads. Both the number of blocks in the grid and the block size are passed along as additional arguments to the kernel invocation, these can be either single values or 3-D tuples.

Through the threads made available on the the device, an architecture known as SIMT - Single Instruction Multiple Threads is be employed (Nvidia, 2019d). Take for example the case of vector addition where you want to add vector A and vector B in order to obtain the resultant output vector. A common solution to this is to iterate through the vector using some kind of loop. However using the CUDA runtime, the same could be achieved by the following source code:

```
#include "cuda_runtime.h"
#include <iostream>
#define N 320


__global__ void add_vectors(int *input_a, int *input_b, int *output){
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    output[i] = input_a[i] + input_b[i];
}

int main(){
    int *dev_a,*dev_b,*dev_out;
    int *host_a,*host_b,*host_out;
    host_a = new int[N];
    host_b = new int[N];
    host_out = new int[N];

    size_t size = N*sizeof(int);
    cudaMalloc(&dev_a,size);
    cudaMalloc(&dev_b,size);
    cudaMalloc(&dev_out,size);
```

Figure 2.8: SM layout of the Pascal architecture. (Nvidia, 2016)

Figure 2.9: Block diagram of the GP-104 with the Pascal architecture (Nvidia, 2016)

```
    for(int i = 0;i<N;i++){
        host_a[i] = 1;
        host_b[i] = 1;
        host_out[i] = 0;
    }
    cudaMemcpy(dev_a,host_a,size,cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b,host_b,size,cudaMemcpyHostToDevice);
    cudaMemcpy(dev_out,host_out,size,cudaMemcpyHostToDevice);
    add_vectors<<<10,32>>>(dev_a,dev_b,dev_out);
    cudaMemcpy(host_out,dev_out,size,cudaMemcpyDeviceToHost);
    int total = 0;
    for(int i = 0;i<N;i++){
        total += host_out[i];
    }
    std::cout<<total<<std::endl;
    delete [] host_a;
    delete [] host_b;
    delete [] host_out;
    cudaFree((void**)dev_a);
    cudaFree((void**)dev_b);
    cudaFree((void**)dev_out);
    return 0;
}
```

Note how the add vectors exposes the current thread index and use it as an index for accessing elements in the different vectors. This way, each thread does exactly one addition. In this program we launch a totalt of 320 threads, using 10 blocks with 32 threads each.

2.2.4 Execution Model

When a kernel is invoked, the thread blocks of the kernel grid are distributed among the SMs. This means that the threads of the block are guaranteed to be executed on the same SM. The threads of the block are grouped together in groups of 32 threads called *warps*. During every cycle on the device, instructions of the kernel are scheduled to eligible warps in the block. All active threads in the warp will then perform the same instruction. For a warp to be eligible, it must have active threads whose instructions are ready to be executed. If a warp is not eligible, it is said to have *stalled*. If this is is the case, the warp scheduler will select another eligible warp from the block that is ready to execute.

As mentioned, instructions are issued and executed for active threads on a per-warp basis. If threads within the same warp becomes inactive as a result of divergent execution branches, the branches will execute sequentially. Take for example an if-statement that evaluates to true for the first 16 threads of the warp, and false for the remainder of the warp's threads. Despite not having any instruction to execute, the inactive threads need to

wait for the execution branch inside the if-clause to complete. As a result, any number of divergent code branches within a warp is going to extend the time it takes for a warp to finish by a factor of the number of divergent branches (Wilt, 2013).

2.2.5 Memory

There are several types of memory available on the SM, each serving a specific purpose. These are the register file, shared memory, L1 cache and texture cache. In addition to the memory on board the SM, there also exists types of memory that are shared between SMs. These are the global memory and L2 cache.

The register file contains the values declared inside the kernel function. In some cases the kernel requires more register memory than the register file can provide. If this is the case, registers are freed by writing their contents to the L1 cache. However, the L1 cache employs a first-in-first-out strategy. This means that the L1 contents could be evicted to L2 cache and then to global memory (Micikevicius, 2017). This method of freeing up registers is known as register spilling.

Shared memory is allocated per block. When shared memory is allocated, every thread within the block is capable of reading and writing to the same area of memory. This allows shared memory to be used for communication between threads. It is about ten times slower for a thread to access its shared memory, compared to accessing its registers. However, accessing global memory is about ten time as slow as shared memory (Wilt, 2013). Because of this, shared memory is often used as a manually controlled cache. To provide a high bandwith, the shared memory is organized into several *banks*. Each of the banks can be accessed simultaneously. If different addresses within the same bank are accessed at the same time, the accesses are serialized (Nvidia, 2019e).

Global memory is the largest memory resource available, and also the one with the highest latency, akin to the RAM of traditional homogeneous CPU systems. When performing operations on global memory the application should strive to make those in a *coalesced* fashion. Memory operations are coalesced on a per- warp basis given that the following criteria are met. The words that are written must be at least 32 bits in size, their addresses contiguous and increasing. Also,the address being accessed by the first thread in each warp must be aligned, with the alignment size itself being dependent on the word size (Wilt, 2013). When coalescence occurs, the compiler will merge multiple memory requests from different threads of the warp into a single request (Nvidia, 2019d). Seeing as accessing global memory takes a relatively high number of cycles, few global memory transactions are desirable.

2.2.6 Occupancy and Throughput

While the GPU has the ability to process a large number of elements in parallel, any one thread could be executed significantly faster on the CPU. When a warp is issued an instruction , a certain number of cycles is going to be needed for the instruction to complete.

These cycles are known as the *latency*. Finding ways of reducing the impact of the latency is known as *hiding* the latency. In practice this is achieved by supplying enough instructions to the warp schedulers such that a new instruction can be issued for every cycle during the latency. This way, instructions can run concurrently and the different units in the SM are saturated.

A contributing factor to throughput, is occupancy. Since the resources of an SM are limited, there can only be certain number of active warps at any give time on each SM. Occupancy can be described as the ratio of active warps to the number of potentially active warps.

$$Occupancy = \frac{Warps\ per\ SM}{Max.\ Warps\ per\ SM} \tag{2.11}$$

There are several factors that can limit the occupancy of a kernel, such as the amount of shared memory used per block, registers used per thread and so forth.

While occupancy is an important aspect when it comes to maximizing throughput, several other strategies such as instruction level parallelism can be leveraged in order to hide the latency caused by both arithmetic computation as well as memory operations. (Volkov, 2010) shows how algorithms that reduces occupancy can, in some cases, more than double the throughput. This illustrates that doing more work per thread can in some cases be a viable strategy.

Chapter 3

Related Work

Previous work in both academic settings and in the open-source community has resulted in several GPU implementations of 3D feature descriptors. Some descriptors even having multiple, independent, GPU implementations. In an effort to address the research questions, previous work has been examined in detail. The implementations whose source code was available to the author, have been given a thorough description in section 3.1. In addition to these, several implementations are mentioned in literature. The core ideas of these are outlined in section 3.2.

3.1 Implementations

3.1.1 Point Cloud Library Implementations

The Point Cloud Library (PCL) is an open-source library for processing 3D data (Rusu and Cousins, 2011). It offers CPU implementations of all the descriptors examined in this thesis. In addition, it also offers GPU implementations of PFH, FPFH, and SI. These implementations have been written using CUDA. (Hu and Nooshabadi, 2015) claims to have implemented the SHOT descriptor on the GPU, and integrated it into PCL. This is not the case at the timne of writing.

All of the PCL implementations share a common requirement for nearest neighbor searching. The CPU implementations by default first constructs a k-d tree before any feature descriptors are computed. During each descriptor computation , this k-d tree is used to search for nearby neighbors. The GPU implementations on the other hand uses an Octree. The octree is implemented on the GPU using CUDA. Building the octree and searching for neighbors is done by two separate kernels.

The first kernel, responsible for building the octree, takes the surface vertices as an input. The second kernel, which searches for neighbors, requires an array of size n where each value represents the index of a feature vertex. In addition, a search radius r and a maximum number of neighbors m must be specified.

There are three outputs of the search. The first is an array of size *Number of feature vertices* . Each value represents the number of neighbors found for the respective feature vertex. The second array is of size $n \cdot m$. Each feature vertex is assigned to an interval of size m. Every value within the interval maps to indices in the surface array. This way the neighborhood of a feature vertex is determined prior to the computation of its descriptor.

As an example, a neighbor search is done specifying 10 as a maximum neighborhood size. The neighbors array has a value of 7 at index 42. This means that the vertex at index 7 in the surface array is a neighbor of the the fifth feature vertex. The third output is m.

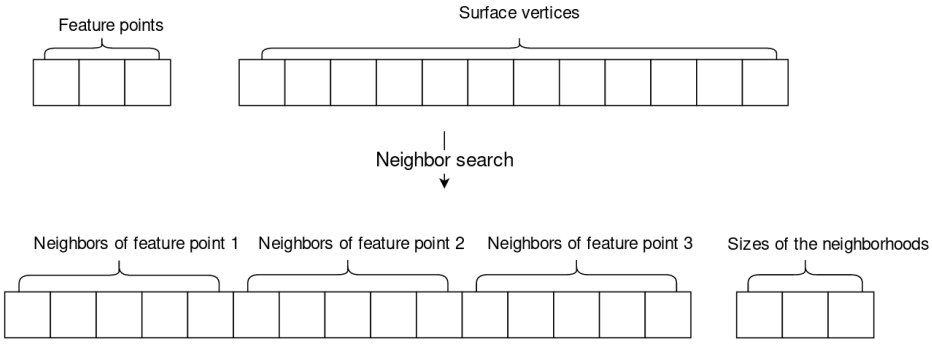


Figure 3.1: The structure of input and output elements in a GPU Octree search

3.1.2 Point Feature Histograms on the CPU

Upon inspection there are some differences between the implementation provided by PCL and of that described in (Rusu et al., 2008). Notably the exclusion of length as one of the functions for computing the pair feature, as well as the usage of five subdivisions instead of two. As a consequence of these adjustments, each PFH now has $5^3 = 125$ bins. (Rusu et al., 2009) argues that the exclusion of length as one of the feature functions has little impact on the robustness of the descriptor.

When computing the features of the vertex pairs, these are stored in a key-value map. Here the key is the combination of vertex indices, and the value is the features of the vertex pair. This effectively caches vertex pairs and reduces the computation required. The motivation behind caching is that feature vertices which are neighbors, are also likely to have a number of other common neighbors. This is illustrated in figure 3.2. The caching scheme is presented in (Rusu et al., 2009).

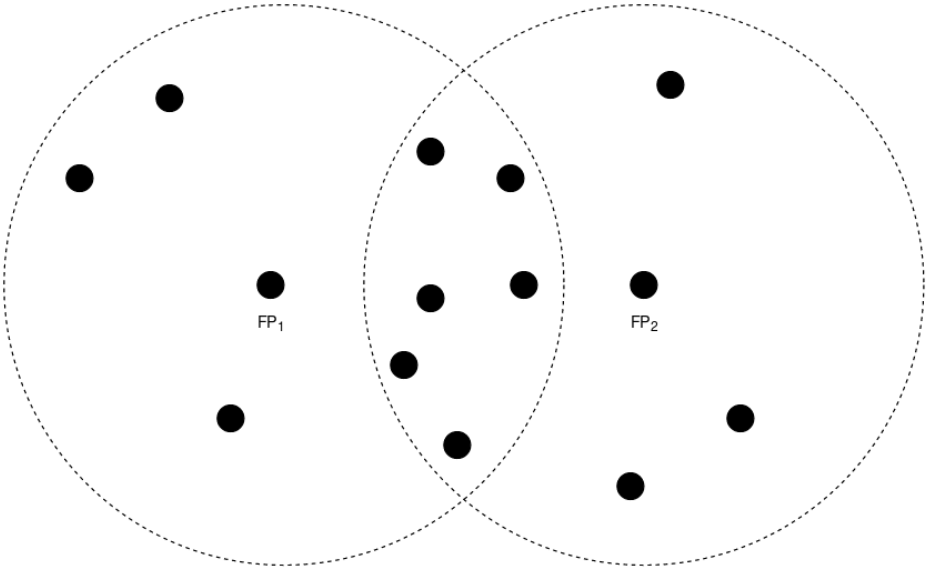


Figure 3.2: Feature vertices sharing neighbors

3.1.3 Point feature histogram on the GPU

Before computation of the PFH descriptors can take place, all feature vertices, neighbor information, surface vertices and their associated normals are transferred to the device from the host. Once the transfer completes the first kernel is invoked.

Repack Kernel

The purpose of the first kernel is to create a 2D array as shown in figure 3.4.

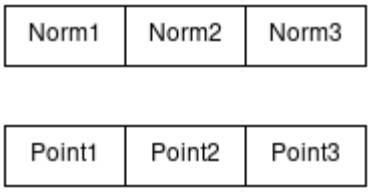


Figure 3.3: Input structure to the repack kernel

Each neighborhood is assigned a warp which is responsible for doing the fetching and restructuring. By default each block contains 256 threads, thus giving a total of 8 warps

Point1.x	Point2.x	Point3.x
Point1.y	Point2.y	Point3.y
Point1.z	Point2.z	Point3.z
Norm1.x	Norm2.x	Norm3.x
Norm1.y	Norm2.y	Norm3.y
Norm1.z	Norm2.z	Norm3.z

Figure 3.4: Output structure to the repack kernel

per block. The block therefore needs to complete the restructuring of 8 neighborhoods. A mathematical definition of the launch size is given by equation3.1.

$$\begin{aligned} \textit{Warps per Block} &= \frac{\textit{Block Size}}{32} \\ \textit{Grid Size} &= \frac{\textit{Number of Feature Vertices}}{\textit{Warps per Block}} \end{aligned} \tag{3.1}$$

Each warp uses its global warp index to access a corresponding interval of the neighbors array. The size of this interval is supplied when invoking the nearest neighbor search kernel earlier, and passed along as a member of its output structure. Each thread in the warp then reads a neighbor element offset by the thread's lane (the thread's position within the warp).

The neighbor element contains an index, which corresponds to a surface vertex element and a normal element. By using this index the thread retrieves the vertex and normal value, assemble them into the 2D format, and stores them in the output.

For example, lane l of the warp with global warp index Idx is going to read the value i stored at index $Idx \cdot \textit{Max Elems} + l$ in the neighbors array. Once retrieved, the thread stores the values in each respective output array (the 2D output contains 6 arrays) at index $Idx \cdot \textit{Max Elems} + l$. In the event that there are more than 32 neighbors, the process is repeated as many times as needed. For every repetition, l is incremented by the warp size. The value stored at index Idx in the neighborhood size array, represents the number of neighbors n that should be processed by the warp. Any thread that does not satisfy the condition $l < n$ is exited. The restructure process is illustrated in figure 3.5.

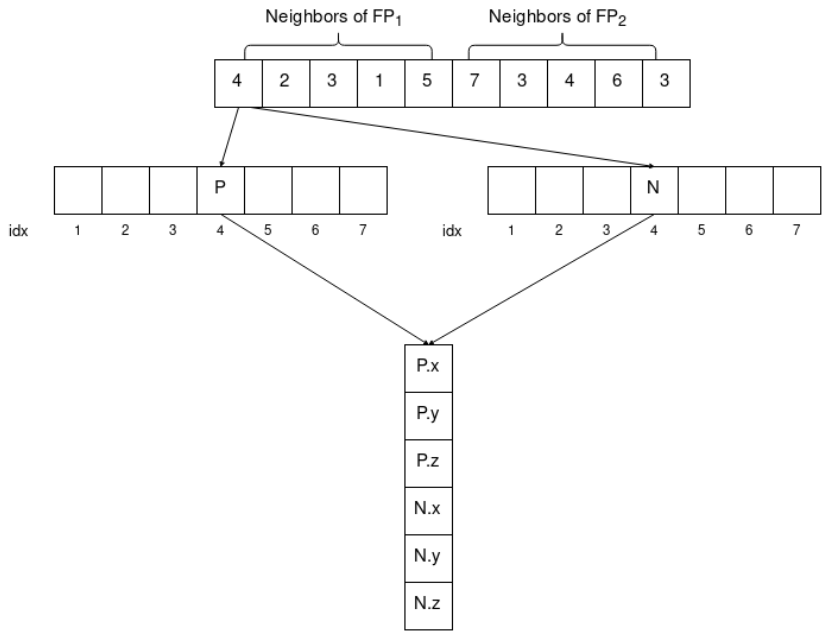


Figure 3.5: Restructure of vertex data

PFH Kernel

The PFH kernel is launched with a block assigned to each feature vertex. Each block contains 256 threads. There are three inputs to the kernel. The first input to the kernel is a pointer to the 2D array created by the previous kernel. The second is a pointer to the array containing neighborhood sizes, created by the neighbor search kernel. The third is the maximum neighborhood size.

Each thread looks up the size s of the neighborhood for the block's respective feature vertex. Afterwards every thread initializes a value $l = \textit{threadIdx.x}$. Following this there is a loop with the constraint that $l < s^2$. In every iteration of the loop, a vertex pair is processed. The indices used to select the pair is given in equation 3.2

$$\begin{aligned} shift &= \textit{Block Index} \cdot \textit{Maximum Neighborhood Size} \\ i &= \left\lfloor \frac{l}{s} \right\rfloor + shift \\ j &= l \mod s + shift \end{aligned} \tag{3.2}$$

The values of the coordinates and normals of vertex i and j are then retrieved from the 2D array created by the repack kernel. Using these values, the same pair features are computed as in the CPU version. From these features the thread computes the correct bin index.

This index is used to increment an array allocated in shared memory representing 8 different histograms (one for each warp of the block), representing the descriptors computed by the block. Each value in this array represents a bin in the histogram. When every pair has been processed, the histogram is written to global memory.

3.1.4 Fast point feature histograms on the CPU

As mentioned previously in section 2.7, computing FPFH descriptors is a two step process. First the SPFH histograms of all vertices in the input surface is created. The PCL implementation does this by performing a nearest neighbor search for every surface vertex, it then computes the vertex's respective SPFH. This is stored in an array of SPFH descriptors. In addition, a lookup table is created such that each vertex in the input surface references a SPFH descriptor.

The next step is to compute the FPFH descriptor. To do this a nearest neighbor search is performed for every vertex considered a feature vertex. Once the neighborhood is determined, the SPFH descriptors of the neighbors is looked up, weighted by the distance to the feature vertex, and summed together. This process computes the final FPFH descriptor. One distinction from the algorithm described in (Rusu et al., 2009) is that the SPFH descriptor of the feature vertex is omitted from the computation of the feature vertex's descriptor such that

$$FPFH(p_q) = \frac{1}{k}\sum_{i=1}^{k}\frac{1}{w_k}SPFH(p_k) \tag{3.3}$$

By default the algorithm is configured to use 11 subdivisions for each of the 3 features, resulting in a histogram size of 33.

3.1.5 Fast point feature histograms on the GPU

The implementation consists of two kernels, one creating the initial SPFH histograms, and the other one creating the final FPFH histogram. Unlike the GPU implementation of the PFH descriptor, there is no kernel responsible for restructuring.

SPFH Kernel

The SPFH kernel is launched with the same grid configuration as the repack kernel described in 3.1.3. Thus its launch parameters are given by equation 3.1. Every warp within the block is responsible for computing a SPFH descriptor of a feature vertex i. Each warp starts by retrieving the coordinates and normal of the vertex i considered to be the feature vertex. This is done by using the warp's global id as an index. The coordinate and normal values are stored in shared memory for later use.

Next, the global warp index idx is used to reference an interval of size m in the neighbours array, and retrieve the size s of the warp's neighborhood.

To compute the feature of a vertex pair, every lane l of the warp looks up the value j stored at index $idx \cdot m + l$ in the neighbors array. The lane then retrieves the coordinates and normals of vertex j and forms a pair with vertex i. Pair features are created as described by section 2.1.4. Each feature is used to increment values of an array stored in shared memory. This array represents w different SPFH descriptors, where w is the number of warps per block. Each thread increments its l value by the warp size and checks to see if the condition $l < s$ is satisfied. The thread will fetch neighbors, compute pair features, increment the histogram and increment l, then check the same condition again, as long this condition is met.

After all SPFH histograms of the block have been computed they are written to global memory. The final format is 2D array where each row correspond to a SPFH descriptor. The vertex of index i will have its SPFH descriptor in row i.

FPFH kernel

The grid configuration of the FPFH kernel is identical to that of the SPFH kernel. To be able to look up neighbors of feature vertices, the outputs of the nearest neighbor search is supplied, as well as the surface vertices. Another input is the 2D array containing SPFH descriptors from the previous kernel.

The warp retrieves the coordinates of the feature vetex i and its neighborhood size. The warp then uses the neighbors array to look up values which maps to corresponding rows in the 2D-array of the SPFH kernel. As mentioned earlier, each row in the array represents a SPFH descriptor. The warp weights each of these descriptors using the squared distance between the feature vertex and the neighbor, and sums them together in a temporary array allocated in shared memory. When all SPFH descriptors have been weighted and added to the temporary array, the array is written to the output residing in global memory. The output structure is a 2D array similar to the one from the SPFH kernel.

3.1.6 Spin Image on the CPU

The SI CPU implementation works by creating a spin image for every feature vertex submitted. In order to create a spin image, a local reference axis needs to be determined. By default the feature vertex's normal will be chosen. The spin image's pixel size is calculated using equation 3.4

$$bin\ size = \frac{\frac{search\ radius}{image\ width}}{\sqrt{2}} \tag{3.4}$$

In order to estimate the neighborhood, a KD-Tree radius search is performed, each of the neighbors are then subjected to spin image mapping. Mapping is done by calculating the α and β coordinates of the current neighbor.

$$\begin{aligned} beta\ bin &= \left\lfloor \frac{beta}{bin\ size} \right\rfloor + image\ width \\ alpha\ bin &= \left\lfloor \frac{alpha}{bin\ size} \right\rfloor \end{aligned} \tag{3.5}$$

Whilst the original description of spin images calls for a different beta bin calculation, the one used for this implementation is justified given that the cylindrical is set to $2 * image\ width$ opposed to the originally proposed height set to $image\ width$. The weights a and b used for biliniear interpolation of the spin image are given by equation 3.6

$$\begin{aligned} a &= \frac{alpha}{bin\ size} - alpha\ bin \\ b &= \frac{beta}{bin\ size} - (beta\ bin - image\ width) \end{aligned} \tag{3.6}$$

Even though they may appear different from the original algorithm, it can be shown that they represent the exact same value. The values of the Spin Image histogram are then incremented similarly to what was described in 2.10.

3.1.7 Spin Images on the GPU

The spin image GPU implementation used in this paper is provided by Bart van Blokland (van Blokland et al., 2018). There are several kernels that takes part in the computation of the spin images, most notable is the kernel used for the actual spin imgage generation. Other utility kernels are also used for initializing histogram values, sampling the mesh, calculating triangle areas and performing the accumulation of these areas.

3.1.8 Utility kernels

The first kernel being launched is to initialize the values of the spin image to zero. Next, the triangle area and cumulative triangle area calculated using equation 3.7

$$\begin{aligned} triangle\ area &= \frac{|side\ one \times side\ two|}{2} \\ cumulative\ area_k &= \sum_{i=1}^{k} triangle\ area_i \end{aligned} \tag{3.7}$$

The goal of the sample kernel is to create a uniform random distribution of points across the model in such a way that the number samples on a triangle is reflected by the triangle's size. In other words, a large triangle should contain a large percentage of the samples, while a small triangle should contain a small percentage of samples.This is achieved through first initializing an array of pairs of random coefficients with size equal to that of the sample size. This is done using the CURAND library.The sample kernel is responsible for creating

the sample points, although first it needs to decide on how many samples there should be for a specific triangle. It does this by looking up the surface area of the triangle and comparing it to the total surface area of the model. The position of a sample point P of a triangle ABC is computed using 3.8 from (Osada et al., 2002).

$$P = (1 - \sqrt{r_1})A + \sqrt{r_1}(1 - r_2)B + \sqrt{r_1}r_2C \tag{3.8}$$

3.1.9 Create descriptors kernel

When launched, the kernel will create the final spin images. In this implementation, the launch parameters for the grid is designed to be a number of blocks equal to the number of vertices in the model, each using 32 threads. At the beginning of the kernel, each thread identifies their block and fetches the corresponding normal and vertex information from global memory. Then every thread in the block is assigned to one of the model's triangles by means of a for-loop which increments by the block size, until every triangle has been iterated. During each iteration the sample bounds of the triangle are calculated, and the corresponding sample points subsequently fetched. For every sample point, $alpha$ and $beta$ are calculated through equation 3.9

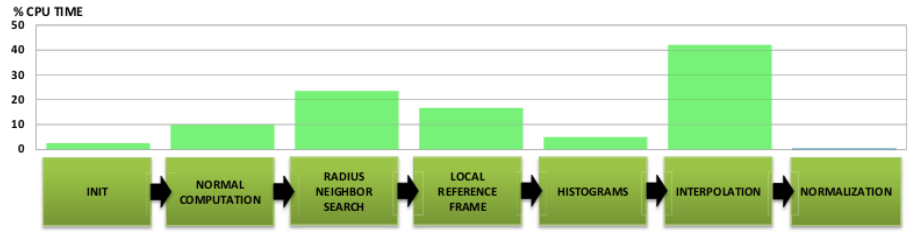
$$\begin{aligned} beta &= \frac{(Sample\ point\ -\ Vertex\ point) \cdot Vertex\ normal}{Vertex\ normal \cdot Vertex\ normal} \\ alpha &= |(Vertex\ point + beta \cdot Vertex\ normal) - Sample\ point| \end{aligned} \tag{3.9}$$

After $alpha$ and $beta$ are obtained, bilinear interpolation can be performed the same way as described in section 2.1.5. In order to avoid race conditions when modifying the histogram values, atomic operations are used for the additions performed by equation 2.10

3.2 Implementations Described In Litterature

3.2.1 SHOT Using OpenCL

(Palossi et al., 2013) describes an implementation of the SHOT descriptor using OpenCL. To create the implementation, the algorithm is broken down into multiple steps. These are extrapolated as kernel functions which are used to form the final SHOT GPU implementation.



.

Figure 3.6: The steps of SHOT algorithm and their contribution to the overall computation on the CPU (Palossi et al., 2013)

Nearest Neighbor Search Kernel

The radius neighbor search identifies neighbour vertices of the feature vertex. A vertex is considered to be a neighbor of the feature vertex if the Euclidean distance d between is less than a specified radius r. The search is done using a kernel with a number of work-groups equal to the number of feature vertices. The number of work-items per work-group is not mentioned in detail. Every work-item then looks up up vertices in the model whose index is a multiple of the work-item's local id.

During each lookup, the vertex's distance d from the feature vertex is calculated. If the vertex is considered a neighbor, its index and distance are stored in arrays residing in private memory. A value in shared memory indicating the number of neighbors found by the work-item is also incremented. The work-item then repeats the process for the next vertex of the surface. After all vertices of the surface have been processed by the work-group, the neighbor indices and their distances are written to global memory.

Local Reference Frame Kernels

To create the LRF there are two kernels involved. The first one creates covariance matrices required. The kernel launches 6 work-groups per feature vertex, one for each unique value

in the covariance matrix of the neighborhood. The final covariance matrix is stored in global memory by reducing the partial results of the work-groups.

The next kernel is responsible for performing Eigenvalue Decomposition and sign disambiguation as described in (Tombari et al., 2010). This kernel is launched with a number of work-groups equal to the number of feature vertices. The Eigenvector computation and sign disambiguation is assigned to a single work-item of the work-group. If the sign disambiguation does not yield a valid result (the sign still being ambiguous), another disambiguation using only 5 neighbors is performed on the host.

Histogram Kernel

The histogram kernel assigns each neighbor vertex to a local histogram based on its location within the spherical support. As mentioned before, the spherical support of the SHOT descriptor has several sub-volumes, each of these have their own local histogram. The kernel uses a grid which assigns a work group to each feature vertex. Each work-item fetches a neighbor vertex of the feature vertex, determines which local histogram it belongs to, and writes the result to global memory.

Interpolation Kernel

The interpolation step is by far the one incurring the highest computational expense as shown in figure 3.6. The kernel launches a grid with a number of work-groups assigned to each feature vertex. Each work-group contains a number of work-items equal to the warp size. The work-items then iterates over the neighbors of the feature vertex.

During each iteration, the coordinates of the neighbor must be fetched. The contribution of the neighbor, in its respective local histogram, is computed. The vertex also contributes to the local histograms of adjacent volumes within the support region. All in all this requires four interpolations to be performed for each neighbor vertex

Normalization Kernel

Finally a kernel performs L_2 normalization of the histograms. The grid used for this contains a work-group for each feature vertex. Each work-item reads a histogram value and stores it in an array in local memory. The array is then reduced, resulting in the normalization value. This value is then used by every work-item to normalize a corresponding histogram bin.

Results

The OpenCL implementation shows an increase in speed for all steps, as illustrated by figure 3.7. Note that the normal computation step is not mentioned in this section. This is because it is particular to point cloud data, whereas the thesis concerns itself with surface meshes.

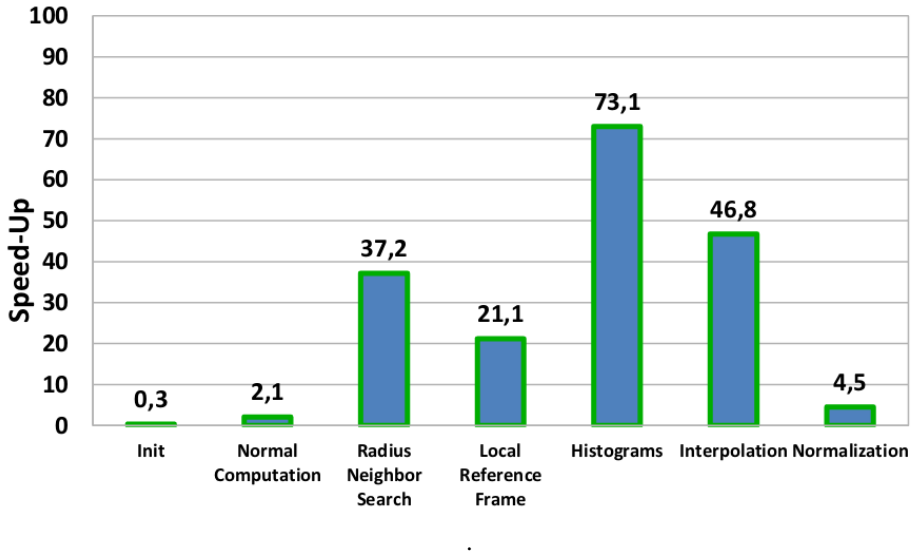


Figure 3.7: The speed-up of the steps involved in computing the SHOT descriptor, when implemented on the GPU. 10 000 descriptors were computed for a model containing 90190 vertices. (Palossi et al., 2013)

3.2.2 FPFH using CUDA

(Garrett et al., 2016) presents a GPU implementation of the FPFH descriptor. Unlike the GPU implementations of the PCL, a KD-Tree is used for nearest neighbor searches. The KD-Tree is also built using CUDA. The output of the neighbor search is similar to the Octree of PCL, however each local neighborhood is cached. This means that the neighbors array contains actual coordinates instead of a value representing a surface index. As with the PCL implementation, SPFH and FPFH computation is split into two kernels.

SPFH Kernel

The SPFH kernel is launched with a block per feature vertex, and block size of k^2 where k is the maximum neighborhood size. This means that each thread is assigned a pair of

neighbors. Each thread retrieves their assigned pair and computes their features. Using these features, the corresponding bins in the SPFH histogram is incremented. The SPFH histogram is located in global memory.

FPFH Kernel

The FPFH kernel launches a grid with a block per feature vertex, and block size of k. Each thread loads a SPFH histogram from global memory and stores a weighted version into shared memory. Once every thread of the block has completed this step, reduction is applied to the values in shared memory. This results in the final FPFH descriptor.

3.2.3 SHOT Descriptor Using CUDA

(Hu and Nooshabadi, 2015) proposes a GPU implementation of the SHOT descriptor. In this approach, there are two kernels. The first kernel computes the LRFs of the feature vertices, while the second kernel maps neighbors to sub-volumes within the spherical support and performs quadrilinear interpolation.

Nearest neighbor searching is done on the host using the KD-Tree implementation of the PCL. In

The LRF kernel assigns a thread to each feature vertex, this ways a single thread computes a single LRF. The number blocks used is given by equation 3.10.

$$\begin{aligned} \textit{block size} &= 256 \\ \textit{number of blocks} &= \frac{\textit{number of feature vertices}}{\textit{number of blocks}} \end{aligned} \tag{3.10}$$

The threads has to iterate through the neighbours of their respective feature vertex, compute the covariance matrix, perform Eigenvalue decomposition and resolve the sign of the Eigenvectors.

Equation 3.10 can also be used to calculate the grid of the histogram kernel. However, the block size used is decreased to 128. As with the previous kernel, each thread is responsible for a single feature vertex. The thread has to determine the local histogram volume, and contribution, of each neighbour.

Chapter 4

Discussion

4.1 Designing a Parallel Feature Descriptor Algorithm For the GPU

4.1.1 Nearest Neighbor Considerations

With the exception of SI, all of the GPU implementations considered for this thesis performs some sort of nearest neighbor search in order to determine support members. All of the PCL implementations used a GPU based octree, whilst (Garrett et al., 2016) used a GPU based k-d-tree. The OpenCL version (Palossi et al., 2013) of SHOT performs a naive nearest neighbour search using the GPU. (Guo et al., 2016) uses the CPU based k-d tree from the PCL.

All of the GPU based nearest neighbor searches stores their result in an array residing in global memory. The total size of this array can be expressed as in equation 4.1 where n is the number of feature descriptors to compute, k is the maximum number of neighbors and s is the element size.

$$\textit{array size} = n \cdot k \cdot s \tag{4.1}$$

All of the implementations that performs a GPU based neighbor search requires the maximum number of neighbors to be specified on kernel invocation. This is likely due to the fact that memory should be allocated to the kernel prior to its launch. While dynamic allocation is possible (Nvidia, 2019a), it is still limited by the heap size. In addition, the expansion and compression of the neighbors array size is likely to add significant complexity when compared to fixed neighborhood sizes.

However, a fixed neighborhood size may impact the final descriptor. This is because there can exist support members that are not evaluated due to the neighbor search already having found the specified maximum amount of neighbors. This can to some extent be prevented by increasing the neighborhood size, thus increasing the recall of the neighbor search. Since any increase in neighborhood size increases the overall size of the neighbors array proportionally, a moderate maximum neighborhood size should be chosen.

When computing feature descriptors, the workload associated with each descriptor is directly related to how many neighbors that are contained within the support region. This means that the workload can become uneven when computing descriptors with a varying number of neighbors. (Guo et al., 2016) shows that setting the maximum neighbor count to the median neighbor count (when performing a nearest neighbor search with perfect recall) reduces the descriptiveness of the (SHOT) descriptor by 5% whilst decreasing the computation time by 15%. This was done on a model where the support count for all descriptors had a minimum value of 6, a median value of 90 and maximum value of 1172.

The CPU implementations that have been examined in this thesis all use the k-d tree provided by the PCL. When a descriptor is extracted, the k-d tree is used to find the respective neighbors of the feature point. These are stored in a std::vector, which is automatically resized as needed. This means that the CPU neighbor search can guarantee a result with perfect precision and recall.

4.1.2 Using Lookup Tables

All of the PCL GPU implementations, as well as the OpenCL version of SHOT, stores the result of the nearest neighbor search as values corresponding to indices in the original surface and normal arrays. For a thread to look up any actual values, it must first determine the correct index in the neighbors array, retrieve the value stored, and read the corresponding index from the surface and normals array. Instead of storing a lookup value to a vertex p in the surface input, one can instead copy the coordinate and normal values into a 2d array as illustrated in figure 3.4.

In fact, the repack kernel in section 3.1.3 describes the conversion to this format from a lookup table. The computational complexity of PFH is $O(n \cdot k^2)$ where n is the number of descriptors to be computed, and k is the maximum number of neighbors. For every descriptor, k^2 pairs of vertices must be fetched for computation. If this step used a lookup table, it would thus require $2 \cdot k^2$ coalesced requests to the array containing neighbor indices residing in global memory. Each of theses transactions would be followed up by non-coalesced accesses to the surface array and the normal array, in order to retrieve the actual values. Because of the prior repack kernel , the thread can instead access the actual values directly based on its lane id.

The repack kernel requires $n \cdot k$ coalesced transactions to global memory for fetching neighbor indices, these result in several non-coalesced transactions each in order to retrieve coordinate and normal values of the neighbors. The values can then be stored in global memory by coalesced transactions. In the PFH GPU implementation, the repack kernel ultimately reduces the need for non-coalesced memory transactions.

Another approach is to modify the nearest neighbor search. Since the coordinates must be fetched in order to check if a vertex is a neighbor, one could store the coordinates directly instead of storing the vertex index. By modifying the nearest neighbor algorithm, one could remove the need for the repack kernel in its entirety. In fact, this design would benefit all GPU based feature descriptor implementations examined in this thesis, that performs nearest neighbor searching. This is a result of no longer needing to perform non-coalesced accesses to neighbor values via a lookup table.

4.2 Optimizing for the GPU

4.2.1 The Importance of Coalescing

Whenever reading or writing to global memory, the threads within the same warp should access memory in such a way that their access pattern is aligned as illustrated in figure 4.1. This will make the compiler try to service multiple memory requests from the same warp

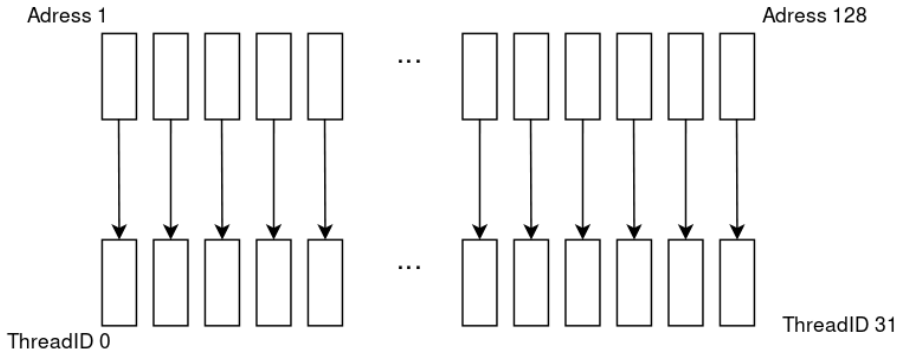


Figure 4.1: Each thread accesses contiguous 4 byte words in global memory

using as few memory transactions as possible. A single memory transaction is capable of transferring 32,64 or 128 bytes of contiguous data to and from global memory (Nvidia, 2019c). This means that given an optimal access pattern, a warp where each thread requests a float (or other 4 byte word) would require a single memory transaction. Figure 4.2 illustrates a non-contiguous access pattern. Since the difference in memory locations accessed by the threads of the warp is greater than what can be covered by a single transaction, multiple transactions will be required. This particular example will require two memory transactions. Even more transactions will be required as the stride between access locations increase. Eventually, each of the threads' memory request would result in their own transaction.

Because of this, any feature descriptor implementations for the GPU should take care to

coalesce any requests to global memory. Examples of such requests are reading normal values, coordinates or writing the computed descriptor to global memory. The result of performing any of these steps in a non-coalesced fashion is that the effective bandwidth becomes severely reduced. The GPU implementations that have been examined for this thesis all employ strategies to ensure coalescence occurs when possible.

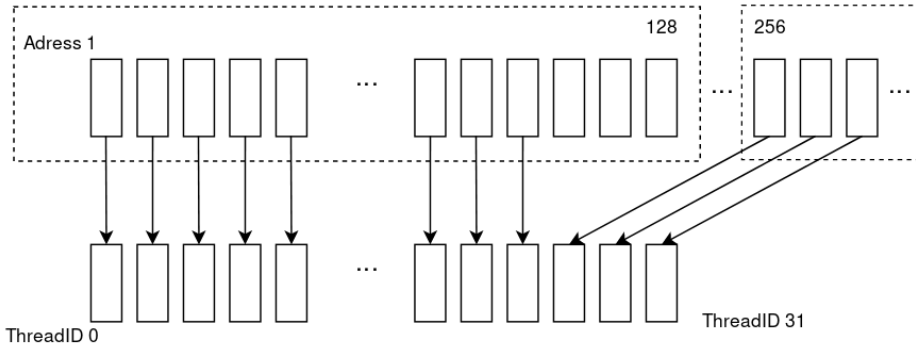


Figure 4.2: Each thread accesses non-contiguous 4 byte words in global memory. This requires two separate fetch instructions to be issued. The data pertaining to each respective transaction is marked with a dotted line.

4.2.2 Avoiding Register Spilling

(Hu and Nooshabadi, 2015) cites register spilling as one of the limiting factors for their computation of the SHOT descriptor using CUDA. The performance can be shown to decrease as much as 38% with 116 bytes being used by local memory (Wilt, 2013). This is due to local memory residing in global memory, which requires a high latency memory transactions.

There are several ways to decrease the requirement for local memory. Devices of compute capability 3.2 and greater can use a maximum of 255 registers per thread.The amount of registers per block varies between 32K and 64K depending on the exact compute capability (Nvidia, 2019b). Evidently, a block of max size (1024 threads) cannot allocate 255 registers to each thread. The block size must therefore be reduced in order to increase the amount of available registers per thread. Failure to so will result in a decrease in performance from spillage to local memory.

Another approach is using non-caching loads when reading data from global memory. From compute capability 5 and greater, data that the compiler identifies as read only, will be cached in L1 (Nvidia, 2019b). This means that global memory reads and registers will contend for cache resources. It is not a given that non-cached loads will increase the overall performance, as cache utilization is going to depend on the program being executed. Depending on the architecture of the device, L1 cache and shared memory may or may not reside on the same physical unit of memory (Nvidia, 2018)(Nvidia, 2016). The CUDA runtime enables developers to specify if a larger L1 cache is desirable, at the expense of a smaller shared memory. The increased size in L1 cache could help alleviate register spilling ,although it might also lead to reduced occupancy and throughput, due to the smaller amount of shared memory available to each thread block. Therefore, this optimization technique and its performance impact, like non-caching loads, depends on the algorithm as a whole.

4.2.3 Optimal Launch Parameters

The launch parameters for kernels should not be overlooked when developing CUDA applications. The size of the grid and its blocks is directly related to the throughput of the application. The examined GPU feature descriptors have used a number of different configurations for their kernel. A general rule of thumb appears to be that the size of a thread block is inversely proportional to the complexity of its respective kernel. In this context, the complexity refers to the amount of resources that must be allocated for each thread block and the number of instructions each thread warp performs.

The GPU implementations from the PCL all assigns a warp of threads to the computation of each descriptor. These warps are then grouped into blocks of 8 warps (256 threads per block). The amount of blocks that fit on each SM depends on the resources that must be allocated for each block, such as shared memory, the number of threads and their use of registers. Each block stays allocated on the device for as long as there are warps containing threads that have not finished their execution.

This means that potentially inactive warps might take up resources that could've been utilized. Since the amount of neighbors for each computed feature descriptor varies (**?**), the warps might end up with relatively uneven workloads. For example, assume that 7 out of 8 warps process sparse neighborhoods and thus finish quickly. This means that the processing of the warp 8 keeps the block allocated, with a smaller number of warps potentially eligible for execution.

(Garrett et al., 2016) uses a block size of k^2 and k for their SPFH and FPFH kernel (where k is the maximum number of neighbors for each descriptor), respectively. This means that the block size, and resources available for each thread will vary depending on what value is chosen for k. When considering the variable neighborhood size, it becomes clear that many of the threads in the block might become redundant when choosing inappropriate values of k. This leads to the same problems as discussed earlier in this section.

Some of the GPU implementations, namely SI and the OpenCL version of SHOT, use a block size equal to that of the warp size for select kernels. The limiting factors are said to be transactions to global memory and register spilling, respectively (van Blokland et al., 2018) (Palossi et al., 2013) .

4.2.4 Atomic Operations and Shared Memory

Shared memory has several use cases when computing feature descriptors. Since shared memory is located on aboard the SM,its access time is lower than that of global memory. Further more, threads within the same block can access the same locations in shared memory. Shared memory is controlled explicitly by the programmer, which means that it will not be spilled to registers. The FPFH implementation of the PCL stores the values of the feature point when executing the FPFH kernel. This acts a caching mechanism when iterating over the neighbors, preventing the values of the feature point from being spilled to global memory.

Histogram computation in parallel requires that the write operations are atomic. This means that when two threads tries to modify the same data, the writes should not overwrite each other. Instead, the write operations should be performed in sequence. The default result of two CUDA threads writing to the same memory location during the same warp is undefined. Starting with the Maxwell architecture, CUDA introduced native shared memory atomics (Nvidia, 2015).

All of the feature descriptors examined outputs a histogram implemented as an array, with histogram bins corresponding to array indices. Since the bin to increment must be computed, and most likely scattered, transactions modifying the histogram are not going to be coalesced. A workaround is to buffer the histogram in shared memory, and use its atomic operations to modify its bins. Once the histogram has finished computing, it can be written to global memory using coalescent transactions. This is illustrated by figure 4.3. A different algorithm has been shown to increase its computation time by up to 200% using this technique. This algorithm created a 2-D histogram based on RGB color channel values in images (Nvidia, 2015).

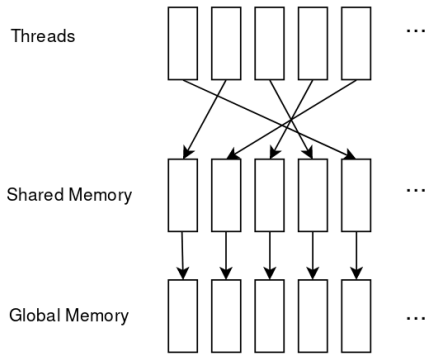


Figure 4.3: Values are written to scattered histogram bins. The threads of the warp then transfer the output to global memory using a coalesced access pattern.

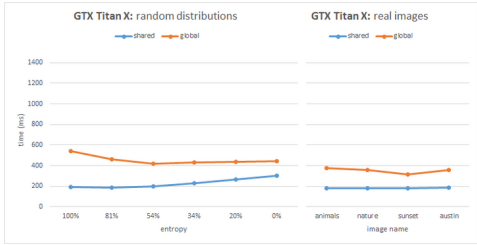


Figure 4.4: The exact speedup with respect to entropy of the data (Nvidia, 2015)

Chapter 5

Conclusion

This section presents a conclusion to the research questions stated in section 1.1. The background for the the answers provided is the accumulated knowledge and observations made from the previous chapters of the thesis.

During this thesis the theory and GPU algorithms of several implementations have been studied in detail. Each of the steps that goes into the different feature descriptors have been examined. It turns out that the different feature descriptors share several steps in their computation. Some examples of such steps are nearest neighbor search, interpolation and histogram computation. Based on the approaches in different implementations, the thesis suggests which ones are likely to give the best computational performance.

When programming for the GPU there are several constraints which, if not considered, may decrease overall performance. The exact impact of these have been shown in other algorithms. The thesis has observed which of these are most relevant when implementing feature descriptors, and suggests which precautions to take in order to ensure that implementations are optimized for GPU execution.

Bibliography

Bro, R., Acar, E., Kolda, T. G., 2008. Resolving the sign ambiguity in the singular value decomposition. Journal of Chemometrics: A Journal of the Chemometrics Society 22 (2), 135–140.

Elseberg, J., Magnenat, S., Siegwart, R., Nüchter, A., 2012. Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. Journal of Software Engineering for Robotics 3 (1), 2–12.

Garrett, T., Radkowski, R., Sheaffer, J., 2016. Gpu-accelerated descriptor extraction process for 3d registration in augmented reality. In: 2016 23rd International Conference on Pattern Recognition (ICPR). IEEE, pp. 3085–3090.

Guo, Y., Bennamoun, M., Sohel, F., Lu, M., Wan, J., Kwok, N. M., Jan 2016. A comprehensive performance evaluation of 3d local feature descriptors. International Journal of Computer Vision 116 (1), 66–89.
URL https://doi.org/10.1007/s11263-015-0824-y

Hebert, M., Johnson, A. E., 05 1999. Using spin images for efficient object recognition in cluttered 3d scenes. IEEE Transactions on Pattern Analysis Machine Intelligence 21, 433–449.
URL doi.ieeecomputersociety.org/10.1109/34.765655

Hu, L., Nooshabadi, S., 2015. G-shot: Gpu accelerated 3d local descriptor for surface matching. Journal of Visual Communication and Image Representation 30, 343–349.

Kammerl, J., Blodow, N., Rusu, R. B., Gedikli, S., Beetz, M., Steinbach, E. G., 2012. Real-time compression of point cloud streams. 2012 IEEE International Conference on Robotics and Automation, 778–785.

Kronos Group, 2019. Opencl overview.
URL https://www.khronos.org/opencl/

Micikevicius, P., 2017. Local memory and register spilling.

URL https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf

Nvidia, 2015. Gpu pro tip: Fast histograms using shared atomics on maxwell.
URL https://devblogs.nvidia.com/gpu-pro-tip-fast-histograms-using-shar

Nvidia, 2016. Whitepaper nvidia geforce gtx 1080 gaming perfected.
URL https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf

Nvidia, 2018. Nvidia turing architecture.
URL https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

Nvidia, 2019a. B.21. dynamic global memory allocation and operations.
URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#dynamic-global-memory-allocation-and-operations

Nvidia, 2 2019b. Compute capabilities.
URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

Nvidia, 2 2019c. Device memory accesses.
URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses

Nvidia, 2 2019d. Nvidia.
URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Nvidia, 4 2019e. Shared memory and memory banks.
URL https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#shared-memory

Nylons, L., 2007. Fast n-body simulation with cuda.

Osada, R., Funkhouser, T., Chazelle, B., Dobkin, D., Oct. 2002. Shape distributions. ACM Trans. Graph. 21 (4), 807–832.
URL http://doi.acm.org/10.1145/571647.571648

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., Phillips, J. C., 2008. Gpu computing.

Palossi, D., Tombari, F., Salti, S., Ruggiero, M., Di Stefano, L., Benini, L., June 2013. Gpu-shot: Parallel optimization for real-time 3d local description. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops.

Rusu, R. B., Blodow, N., Beetz, M., 2009. Fast point feature histograms (fpfh) for 3d registration. In: 2009 IEEE International Conference on Robotics and Automation. IEEE, pp. 3212–3217.

Rusu, R. B., Blodow, N., Marton, Z. C., Beetz, M., 2008. Aligning point cloud views using persistent feature histograms. In: 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, pp. 3384–3391.

Rusu, R. B., Cousins, S., 2011. Point cloud library (pcl). In: 2011 IEEE international conference on robotics and automation. pp. 1–4.

Silpa-Anan, C., Hartley, R., 2008. Optimised kd-trees for fast image descriptor matching. In: 2008 IEEE Conference on Computer Vision and Pattern Recognition. IEEE, pp. 1–8.

Tombari, F., Salti, S., Di Stefano, L., 2010. Unique signatures of histograms for local surface description. In: European conference on computer vision. Springer, pp. 356–369.

van Blokland, B. I., Theoharis, T., Elster, A. C., 2018. Quasi spin images.

Volkov, V., 9 2010. Better performance at lower occupancy.
URL https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf

Wilt, N., 2013. The CUDA Handbook: A Comprehensive Guide to GPU Programming. Addison-Wesley.
URL https://books.google.no/books?id=KUxsAQAAQBAJ

Appendix